SEDAR: Detectando y Recuperando Fallos Transitorios en Aplicaciones de HPC

Diego Montezanti[1,4], Enzo Rucci[1], Dolores Rexachs[2],
Emilio Luque[2], Marcelo Naiouf[1] y Armando De Giusti[1,3],

[1] III-LIDI, Facultad de Informática, UNLP. CEA CICPBA.
Calle 50 y 120, 1900 La Plata (Buenos Aires), Argentina
{dmontezanti, erucci, mnaiouf, degiusti}@lidi.info.unlp.edu.ar
[2] Departamento de Arquitectura de Computadoras y Sistemas Operativos, UAB
Campus UAB, Edifici Q, 08193 Bellaterra (Barcelona), Spain
{dolores.rexachs, emilio.luque}@uab.es
[3] Consejo Nacional de Investigaciones Científicas y Tecnológicas
[4] Instituto de Ingeniería y Agronomía, UNAJ
Av. Calchaquí 6200, 1888 Florencio Varela (Buenos Aires), Argentina

Resumen. El manejo de fallos es una preocupación creciente en HPC; en el futuro, se esperan mayores variedades y tasas de errores, intervalos de detección más largos y fallos silenciosos. Se proyecta que, en sistemas de exa-escala, los errores ocurran varias veces al día y se propaguen para generar desde caídas de procesos hasta corrupciones de resultados debidas a fallos no detectados. En este trabajo se describe la utilización de SEDAR, una herramienta que permite detectar fallos transitorios en aplicaciones MPI, y recuperar automáticamente las ejecuciones, posibilitando su finalización con resultados fiables. La detección se basa en replicación de procesamiento y monitorización del envío de mensajes y del cómputo local, mientras que la recuperación se logra utilizando múltiples *checkpoints* de capa de sistema. El estudio del comportamiento de SEDAR en presencia de fallos, inyectados en distintos momentos durante la ejecución, permite evaluar su desempeño y caracterizar el *overhead* asociado a su utilización. Las posibilidades de configurar el modo de uso, adaptándolo a los requerimientos de cobertura y máximo *overhead* permitido de un sistema particular, hacen de SEDAR una metodología factible y viable para la tolerancia a fallos transitorios en sistemas de HPC.

Palabras clave: detección de fallos transitorios, recuperación automática, corrupción silenciosa de datos, aplicaciones de HPC, inyección de fallos.

1 Introducción

En el área del Cómputo de Altas Prestaciones (HPC - *High-Performance Computing*), el número de componentes de los sistemas paralelos continúa en aumento, en la búsqueda de mejorar la *performance*, y, como consecuencia, la confiabilidad se ha vuelto un aspecto crítico. En las plataformas actuales, las tasas de fallos son de pocas horas [1], y está previsto que en los sistemas de la exa-escala, las grandes aplicaciones paralelas tengan que lidiar con fallos que ocurran cada pocos minutos [2], por lo que requerirán ayuda externa para progresar eficientemente.

Algunos reportes recientes [3] han señalado a la Corrupción Silenciosa de Datos (SDC) como el tipo de fallo más peligroso que se puede presentar en una aplicación paralela, ya que producen que el programa aparente ejecutarse correctamente, pero finalice con resultados incorrectos. Algunas causas posibles de SDC son la radiación cósmica y la contaminación en los empaquetamientos de los circuitos, entre otros. Pueden afectar a bits de la caché, de la memoria principal o de los registros de la CPU [19]. Las aplicaciones científicas y las simulaciones a gran escala son las áreas más afectadas; en consecuencia, el tratamiento de los errores silenciosos es uno de los mayores desafíos en el camino hacia los sistemas resilientes.

En aplicaciones de paso de mensajes, y sin mecanismos de tolerancia apropiados, un fallo silencioso, que afecte a una única tarea, puede producir profundos efectos de corrupción de datos, causando un patrón de propagación en cascada, a través de los mensajes MPI, hacia todos los procesos que se comunican [4]; en el mejor escenario, los resultados finales erróneos podrán detectarse recién al finalizar la ejecución.

Una forma posible de tolerar estos fallos es diseñar hardware redundante (registros y ALU´s), pero es una solución muy costosa y difícil de implementar [5]. Por lo tanto, se requieren soluciones de software específicas, con una relación costo/beneficio más adecuada, para evitar la necesidad de volver a realizar la ejecución completa en caso de detectar un fallo. Una de ellas es la ejecución redundante, en la cual un proceso es duplicado, y ambas réplicas realizan la misma secuencia de ejecución; en aplicaciones determinísticas, ambas copias producen la misma salida para la misma entrada [6]. En el contexto de HPC, la utilización de arquitecturas *multicore* con este fin representa una solución viable para detectar SDC´s, debido a su redundancia natural [1].

Las técnicas de *Checkpoint & Restart* (C/R) constituyen una alternativa bien conocida para restaurar las ejecuciones de aplicaciones paralelas cuando existen fallos permanentes en los sistemas, que causan la caída de procesos o de nodos de cómputo completos (*fail-stop*) [7]. Los *checkpoints* coordinados, por ejemplo, almacenan periódicamente el estado completo de una aplicación, de forma de que todos los procesos pueden recomenzar desde el último *checkpoint* almacenado si ocurre un fallo. Lamentablemente, las técnicas de C/R introducen un alto *overhead* temporal, que crece a medida que aumenta el tamaño del sistema. Sin embargo, su mayor limitación consiste en que no hay garantía de que un *checkpoint* sea un punto seguro de recuperación, debido a que en su estado pueden haberse almacenado errores no detectados. Si el cómputo está fuertemente acoplado, esta situación empeora, debido a que un error en un nodo puede propagarse a los demás en pocos microsegundos [8].

En este escenario que conjuga resultados no fiables y altos costos de verificación, en los últimos años se ha diseñado la metodología SEDAR (*Soft Error Detection and Automatic Recovery,* anteriormente llamado SMCV [9]) para proveer tolerancia a fallos transitorios a sistemas formados por aplicaciones paralelas que utilizan paso de mensajes y se ejecutan en *clusters* de *multicores*. SEDAR es una solución basada en replicación de procesos y monitorización de los envíos de mensajes y el cómputo local, que aprovecha la redundancia de hardware de los *multicores*, en la búsqueda de brindar ayuda a programadores y usuarios de aplicaciones científicas a obtener ejecuciones confiables. SEDAR proporciona dos variantes para lograr la tolerancia a fallos: un mecanismo de detección y relanzamiento automático desde el comienzo; y un mecanismo de recuperación, también automático, basado en el almacenamiento de múltiples *checkpoints* de nivel de sistema. En este artículo se estudia el

comportamiento de SEDAR en presencia de fallos, inyectados en distintos momentos durante la ejecución, para evaluar su desempeño y caracterizar el *overhead* asociado a su utilización, demostrando así su eficacia y viabilidad para la tolerancia a fallos transitorios en sistemas de HPC.

El resto del artículo se organiza de la siguiente manera. La Sección 2 repasa algunos conceptos básicos y describe trabajos relacionados. La Sección 3 revisa conceptualmente el funcionamiento de las distintas variantes de la metodología propuesta. La Sección 4 describe SEDAR como herramienta, sus modos de uso y los fundamentos de su implementación. En la sección 5 se detallan los experimentos realizados para caracterizar el comportamiento de SEDAR, cuando es incorporado a una aplicación de prueba, en ausencia y en presencia de fallos. Por último, la Sección 6 presenta las conclusiones y las líneas de trabajo futuras.

2 Conceptos Básicos y Trabajos Relacionados

Los fallos transitorios se han clasificado (de acuerdo a sus efectos sobre la ejecución de las aplicaciones) en [9]: Errores Latentes (LE-*Latent Errors*), que afectan datos que no son utilizados posteriormente, no teniendo impacto en los resultados; Errores Detectados Irrecuperables (DUE-*Detected Unrecoverable Errors*), que causan anomalías detectables para el software del sistema, sin posibilidad de recuperación, produciendo que las aplicaciones finalicen abruptamente; Errores por *Time-Out* (TOE), que producen que el programa no finalice dentro de un lapso determinado; y Corrupciones Silenciosas de Datos (SDC-*Silent Data Corruptions*), que no son detectadas por el software del sistema, por lo que sus efectos se propagan para producir la finalización con salidas incorrectas. En aplicaciones paralelas que se comunican, pueden causar: Corrupción de Datos Transmitidos (TDC-*Transmitted Data Corruption*), que afecta a datos que se transmiten en algún momento (por lo que si no se detecta, se propaga hacia otros procesos); o Corrupción de Estado Final (FSC-*Final Status Corruption*), donde los datos alterados no se transmiten, pero se propagan localmente, corrompiendo los resultados finales del proceso afectado.

Para las tecnologías actuales es muy complejo lidiar con SDC frecuentes. Algunas soluciones algorítmicas existentes sólo pueden aplicarse a *kernels* específicos [10], en tanto que las propuestas de detección basadas en el compilador o en software de *runtime* son más generales, pero también significativamente más complejas. Por otra parte, las estrategias de contención buscan mitigar las consecuencias de los fallos, previniendo su propagación hacia otros nodos o hacia los datos almacenados en *checkpoints*, que son necesarios para la recuperación [7]. Los autores de [11] han mostrado que la replicación resulta más eficiente que C/R en contextos de altas tasas de fallo y altos *overheads* de *checkpointing*.

Tradicionalmente, la detección de SDC se ha abordado a través de replicación de procesamiento, combinada con algún grado de comparación de resultados durante la ejecución. Las soluciones basadas en redundancia de software posibilitan prescindir de hardware costoso, realizando la replicación a nivel de *threads* [12], procesos [5] o estado de la máquina. Otras soluciones, como la replicación aproximada [7] involucran menos recursos sacrificando precisión, por ejemplo, estableciendo límites superior e inferior para los resultados del cómputo. MR-MPI [13] propone replicación

parcial, que puede combinarse con C/R en los procesos que no se replican [14], para proporcionar una solución transparente para el ámbito de HPC. rMPI [11] se ocupa de fallos que causan la caída de procesos mediante ejecución redundante de aplicaciones MPI, por lo que el programa falla sólo si lo hacen dos nodos replicados; esta forma de redundancia escala, en el sentido de la probabilidad de que un nodo y su réplica fallen simultáneamente disminuye al aumentar el número de nodos; sin embargo, este beneficio tiene el costo de duplicar la cantidad de recursos y cuadruplicar el número de mensajes.

RedMPI [4] es una librería MPI que utiliza la replicación por proceso de rMPI para detectar y corregir SDC, comparando en el receptor los mensajes enviados por emisores replicados. Utiliza una optimización basada en *hashing* para evitar enviar y comparar los contenidos completos de los mensajes; no requiere cambios al código fuente y garantiza determinismo entre las réplicas. RedMPI puede ser utilizado en sistemas de gran escala, ya que provee protección aún en entornos con altas tasas de fallos. RedMPI permite que las réplicas sean mapeadas en los mismos nodos físicos que los procesos originales; en este sentido, SEDAR se comporta de manera similar. RedMPI también monitorea las comunicaciones, por lo que retrasa la detección hasta la trasmisión de mensajes, pero, a diferencia de SEDAR, la validación se realiza del lado del receptor, lo que produce *overhead*, latencia y tráfico de red adicionales.

En [6] se abordan los fallos en sistemas de memoria compartida, mediante un esquema de múltiples hilos por proceso, que incluye el manejo del no-determinismo debido a los accesos a memoria. En [1] se describe un protocolo para programas híbridos que utilizan tareas y MPI, que permite la recuperación basada en *checkpointing* no coordinados y *logging* de mensajes: únicamente la tarea afectada por el error se reinicia, y todas las llamadas a MPI son manejadas dentro de ella.

También se han desarrollado protocolos tolerantes a fallos para otros modelos de programación paralela, como PGAS [15]. En [2] se explora la combinación de realizar *checkpointing* de los resultados de las tareas, y replicación para detección específica de la aplicación, en un contexto de flujos de trabajo lineales, tanto en presencia de fallos *fail-stop* como silenciosos.

3 Breve Revisión de la Metodología SEDAR

Como se mencionó en la introducción, SEDAR proporciona dos variantes para lograr la tolerancia a fallos: un mecanismo de detección y relanzamiento automático desde el comienzo, y un mecanismo de recuperación basado en el almacenamiento de múltiples *checkpoints* de nivel de sistema. A continuación se revisan conceptualmente ambas estrategias propuestas.

3.1 Detección y Relanzamiento Automático

Para detectar fallos transitorios durante la ejecución de aplicaciones paralelas determinísticas, SEDAR valida los contenidos de los mensajes que se van a enviar, acotando así la latencia de detección, aislando el fallo y evitando que se propague a los demás procesos. Para ello, duplica en un *thread* cada proceso de la aplicación. Al alcanzar el instante de envío de un mensaje, las réplicas utilizan un mecanismo de

sincronización para garantizar que se encuentran en el mismo punto, y allí se comparan los contenidos de los mensajes computados por ambos hilos; si hay coincidencia, sólo una réplica envía el mensaje (bajo la premisa de que no hay fallos en las comunicaciones), no requiriéndose ancho de banda adicional. El proceso receptor espera a que su réplica alcance el mismo punto y le copia los mensajes que ha recibido para luego reanudar la ejecución. Por otro lado, los fallos cuyos efectos no se transmiten a otros procesos, pero que se propagan localmente hasta el final son detectados en una operación de validación de resultados finales. En [9] se estudia compromiso entre la latencia de detección y la sobrecarga computacional involucrada en realizar validaciones más frecuentes.

Este mecanismo posibilita detectar fallos que causan TDC y FSC; además, bajo la premisa de que, en un sistema homogéneo dedicado, los tiempos de ejecución de dos réplicas que realizan el mismo cómputo deben ser similares, es posible detectar los TOE al notar una diferencia considerable entre los tiempos de procesamiento de ambas réplicas [16], a causa posiblemente de un fallo silencioso. La Fig. 1 esquematiza el funcionamiento del mecanismo de detección frente a la ocurrencia de un fallo.

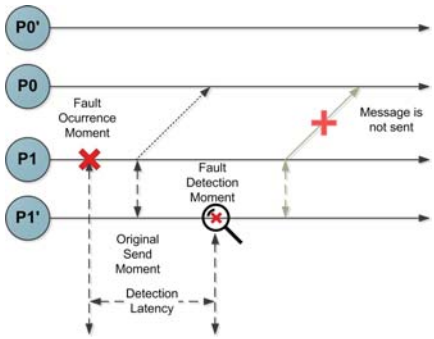


Fig. 1. Operación del mecanismo de detección frente a la ocurrencia de un fallo

El mecanismo de detección propuesto tiene situaciones de vulnerabilidad asociadas, que han sido descriptas previamente en [16]; sin embargo, las situaciones que lo afectan son extremadamente improbables, de modo que pueden ignorarse sin riesgos considerables.

En ausencia de un mecanismo de recuperación, la ocurrencia de un fallo de los tipos mencionados causa que la aplicación detenga su ejecución al momento de la detección, evitando así la costosa e innecesaria espera a que finalice con resultados incorrectos, y permitiendo su relanzamiento inmediato y automático desde el comienzo [17].

3.2 Recuperación Basada en Múltiples *Checkpoints* de Capa de Sistema

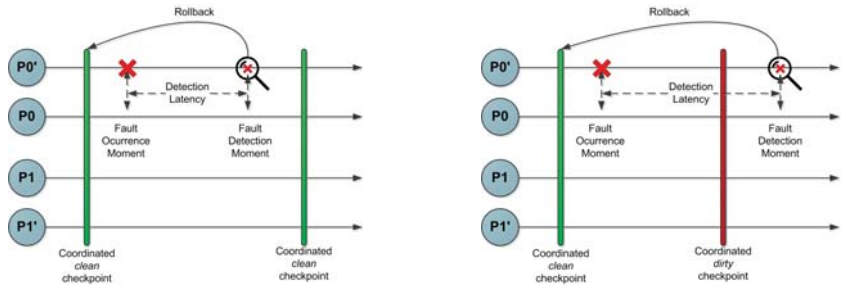
El mecanismo de recuperación de SEDAR se basa en el almacenamiento de una cadena de múltiples *checkpoints* coordinados y distribuidos que se realizan por medio de una librería de *checkpointing* de capa de sistema. El hecho de guardar varios *checkpoints* es necesario debido a que, en el *checkpoint* de nivel de sistema, el usuario no tiene control sobre la información que se guarda. Por lo tanto, no es posible garantizar que un determinado *checkpoint* constituya un punto seguro de recuperación, debido a que los datos pueden haber sido alterados por la ocurrencia de un fallo transitorio antes de ser almacenados. En consecuencia, los *checkpoints* previos al último no pueden ser descartados, y, en el caso de que se detecte un fallo, se debe determinar la factibilidad de recuperar la ejecución desde el último *checkpoint*

(en el mejor caso) o desde alguno anterior [8]. Como casos particulares, si un fallo es detectado cuando aún no se ha almacenado ningún *checkpoint*, la aplicación debe ser relanzada desde el comienzo. En tanto, si en caso de detectar un fallo, se regresa a un *checkpoint* corrupto, en el intento de re-ejecución se vuelve a detectar el mismo fallo y es necesario retroceder hasta uno previo; esta situación puede ocurrir repetidas veces si la latencia de detección es muy alta [17]. En las Fig. 2 (a) y (b) se esquematizan estos comportamientos. El algoritmo de recuperación, en pseudocódigo, se muestra en la Fig. 3.

La automatización de este mecanismo reside en que un proceso externo a la aplicación pueda acceder al valor de `external_counter` y buscar el *script* de reinicio correspondiente (entre todos los generados por la librería de *checkpointing*) de acuerdo con el valor leído.

Este método tiene como principales ventajas la de ser transparente a la aplicación, en el sentido de que no es necesario conocer su estructura interna, ni disponer de *checkpointing* a medida de la aplicación; además, el mismo mecanismo de C/R es el que ya se usa para los fallos permanentes. Por lo tanto, la utilización de un mecanismo de detección basado en duplicación, y uno de recuperación basado en C/R posibilita detectar y corregir fallos silenciosos sin la necesidad de recurrir a la triplicación con mecanismo de votación [20]. En tanto, sus limitaciones consisten, por un lado, en la cantidad de almacenamiento requerido, al no poder eliminar los *checkpoints* previos; por otro, el gasto de tiempo involucrado en los múltiples intentos de reinicio posibles [8]; y, por último, la baja escalabilidad de los *checkpoints* coordinados al crecer la cantidad de procesos, además del hecho de que almacenan gran cantidad de información relacionada al sistema [1].

Cabe aclarar que la metodología SEDAR propone también un mecanismo de recuperación basado en *checkpoints* de capa de aplicación, que disminuye el tiempo y el trabajo asociados a la recuperación a costa de requerir validación de los *checkpoints* [17]



(a) Baja latencia de detección: la recuperación se realiza desde el último *checkpoint*

(b) Alta latencia de detección: la recuperación se realiza desde el *checkpoint* anterior

Fig. 2. Posibles casos de recuperación utilizando múltiples *checkpoints* de nivel de sistema, dependiendo de la latencia de detección

```
// contador externo que controla la cantidad de rollbacks
// (no se almacena en el checkpoint)
int external_counter = 0;
// variable boolean que indica si se detectó una falla en la última ejecución
boolean fault_detected = FALSE;
// ejecutar aplicación monitorizada por SEDAR
SEDAR_exec(app)
// se controla si hubo falla y, en ese caso, se ejecuta el mecanismo de recuperación
while ( fault_detected == TRUE) do
    // se incrementa en uno el contador externo
    external_counter++;
    // obtener cantidad de checkpoints realizados
    num_ckpt = get_checkpoint_count();
    // resetear variable boolean
    fault_detected=FALSE;
    // re-ejecutar aplicación desde el checkpoint num_ckpt - external_counter
    SEDAR_restart(app,num_ckpt - external_counter);
end;
```

Fig. 3. Pseudocódigo del algoritmo de recuperación basado en múltiples *checkpoints* periódicos realizados desde el exterior

4 SEDAR como Herramienta

4.1 Modos de Utilización

En su estado actual, SEDAR permite 3 modos de utilización:

(a) Sólo detección y relanzamiento automático

(b) Recuperación basada en *checkpoints* de nivel de sistema disparados por eventos

(c) Recuperación basada en *checkpoints* periódicos de nivel de sistema

El modo **(a)** permite evitar el coste asociado al almacenamiento de los *checkpoints*, posibilitando que la aplicación pueda ser relanzada desde el principio ni bien transcurre la latencia de detección. Este modo es útil cuando el *overhead* asociado a los *checkpoints* es muy alto, o cuando los sucesivos intentos fallidos de re-ejecución de manera regresiva representan un gasto de tiempo superior al de simplemente relanzar desde el comienzo. En la Sección 5.2 se verá una evaluación de estos costos para un caso de estudio.

El modo **(b)** es de utilidad cuando, en la búsqueda de obtener ejecuciones confiables, se tiene un conocimiento detallado del comportamiento de la aplicación a proteger, por ejemplo, en cuanto a la relación cómputo/comunicaciones. En este caso, SEDAR puede sintonizarse con la aplicación, minimizando el *overhead* introducido, ya que los *checkpoints* pueden sincronizarse con eventos significativos de la ejecución, como pueden ser determinadas fases de comunicación. Tomando en cuenta que los datos a comunicar son validados previamente, almacenar *checkpoints* en los instantes de comunicación permite minimizar los riesgos de almacenar valores corruptos y de propagarlos; esto lo hace un mecanismo muy viable para ser utilizado en aplicaciones tipo SPMD. Por otra parte, seleccionar cuántos *checkpoints* se realizan (y en qué instantes) posibilita optimizar la relación costo/beneficio del mecanismo de protección, y mantener acotado el *overhead*. Este modo fue el utilizado

principalmente en la etapa de validación funcional del mecanismo: basándose en el conocimiento de una aplicación de prueba, y mediante experimentos de inyección controlada de fallos y *checkpoints* sincronizados con las comunicaciones, pudo determinarse, en cada caso, qué dato resultaba afectado, en qué instancia se detectaba ese fallo y desde qué punto podía recuperarse la ejecución [17]. Como desventaja, este mecanismo no es transparente (ni automático), en el sentido de que el código de los *checkpoints* debe insertarse dentro de la aplicación.

Como contrapartida, el modo **(c)** permite proteger a la aplicación "desde el exterior". Lanzando la ejecución bajo la órbita de un proceso coordinador de la librería de *checkpointing*, se establece un intervalo de almacenamiento periódico. En este caso, no es necesario conocer la aplicación en profundidad, a costa de no tener control sobre el instante preciso en que se realiza el *checkpoint*, ni de desde dónde podrá recuperarse la ejecución. El *overhead* tendrá un comportamiento asociado al costo de almacenar un *checkpoint* y a la cantidad de ellos que se almacenen dado el intervalo de periodicidad. Como ventaja, esta protección pude realizarse de forma completamente automática.

4.2 Implementación

El mecanismo de detección se implementa en la forma de una librería que modifica las funciones y los tipos de datos definidos en MPI (sintácticamente, sólo cambia el prefijo MPI por SEDAR), y agrega las funciones `SEDAR_Call` y `SEDAR_Validate`. Por otro lado, utiliza **Pthreads** para la replicación y sincronización de hilos. Para incorporar la detección, el código de la aplicación debe ser levemente modificado y recompilado, en un procedimiento automatizable.

La recuperación automática se implementa mediante la utilización de la librería DMTCP [21] (versión 2.4.4) que provee *checkpointing* coordinado y distribuido de nivel de sistema. El comportamiento por defecto de la librería consiste en almacenar un único *checkpoint*, sobre-escribiendo el archivo generado cada vez que se guarda un *checkpoint* nuevo; durante la instalación, puede modificarse este comportamiento mediante la configuración del *flag* `enable_unique_checkpoint = no`, de modo que se pueden almacenar varios *checkpoints* (con números correlativos) y sus correspondientes *scripts* de reinicio, sin perder ningún estado previo; en esta posibilidad está basado el mecanismo de recuperación de SEDAR. DMTCP genera un proceso coordinador, que es el encargado de monitorear la aplicación objetivo y realizar los *checkpoints* (en este caso periódicos) desde el exterior, lo cual produce que el mecanismo de recuperación sea transparente para la aplicación [17].

5 Trabajo Experimental

5.1 Diseño de la Experimentación

Se diseñaron experimentos destinados a determinar la incidencia de la incorporación de SEDAR a la ejecución de una aplicación paralela que utiliza MPI, mostrando además la eficacia de los mecanismos de detección y de recuperación automática. SEDAR se utilizó tanto en las variantes (a) y (c) descritas en la Sección 4.1. Para ello,

debió modificarse el código fuente de la aplicación para integrarla con la funcionalidad de SEDAR. La aplicación de prueba fue una multiplicación de matrices paralela, cuya implementación sigue el modelo *Master-Worker*, donde uno de los procesos es responsable de la distribución del trabajo y la recuperación de los resultados parciales. Para computar C = A × B, el *Master* envía a cada proceso (inclusive a sí mismo) un bloque de filas de la matriz A y la matriz B en forma completa. Luego, cada proceso es responsable de calcular un bloque de filas de C. Por último, resulta importante mencionar que los procesos emplean operaciones de comunicación colectiva para el intercambio de datos (MPI_Scatter para distribuir A, MPI_Bcast para enviar B y MPI_Gather para recuperar los resultados parciales de C). Debe notarse que esta aplicación demanda gran cantidad de cómputo regular y que requiere poca cantidad de comunicaciones al inicio y al final de cada tarea. Para los experimentos, se utilizaron 8 procesos y matrices de 8192 x 8192 elementos.

Los experimentos consistieron en medir los tiempos de ejecución de ambas variantes de SEDAR en ausencia de fallos, para determinar el *overhead* introducido respecto de: (1) una ejecución sin ninguna protección y (2) una protegida manualmente por medio de ejecución simultánea de dos instancias independientes. Para el caso de la recuperación automática, se buscó establecer la relación entre el *overhead* y el intervalo de *checkpointing* (**i**), midiéndolo para tres valores distintos: **i** = 60s, **i** = 90s, **i** = 120s; para cada caso la cantidad de *checkpoints* almacenados fue **Nckpt** = 6, **Nckpt** = 4 y **Nckpt** = 3, respectivamente. Estos intervalos fueron definidos sobre la base de que la ejecución, sin la incorporación de SEDAR, dura aproximadamente 6 minutos; tomando esto en cuenta, los intervalos seleccionados buscaron ofrecer distintas alternativas de compromiso entre la protección brindada y el *overhead* introducido, frente a distintos valores posibles de MTBF (*Mean Time Between Failures*).

Luego se realizaron experimentos en presencia de fallos, contemplando tres casos de inyección: uno al comienzo de la ejecución, uno aproximadamente a la mitad, y uno hacia el final con la variante de detección de SEDAR. En el caso de la de recuperación, los experimentos para cada inyección se repitieron con cada uno de los valores de intervalo de *checkpoint*. El objetivo de estas pruebas fue determinar el comportamiento de SEDAR en presencia de fallos, y poder obtener conclusiones respecto de la conveniencia de utilizar cada variante, en función del momento de ocurrencia del fallo y el intervalo de *checkpoint* configurado. La información obtenida le permite al usuario configurar SEDAR para funcionar de manera óptima en conjunto con la aplicación monitorizada.

Los experimentos se llevaron a cabo utilizando dos nodos de un *cluster* Blade; cada nodo contiene dos procesadores Intel Xeon e5405 de 4 núcleos a 2.0 GHz, 10 GB de memoria RAM, disco local de 250 GB y una conexión de red Gigabit Ethernet. El sistema operativo es Debian 6.0.7 (64 bits, versión de kernel 2.6.32) y la versión de la librería de comunicaciones es OpenMPI 1.7.5.

5.2 Resultados Experimentales

Los resultados de todas las pruebas realizadas se resumen en la Tabla 1. En todos los casos, los tiempos obtenidos son el promedio de 5 repeticiones de cada experimento particular. El tiempo de la primera fila es el de la ejecución de la

aplicación MPI pura, con la particularidad de que el *mapping* se realizó con 4 procesos en cada nodo, aun quedando *cores* libres, de forma de forzar la utilización de la red, para que la comparación con SEDAR (que replica los procesos) fuese más justa. En la segunda fila, se muestra el tiempo de ejecutar dos instancias independientes simultáneas de la aplicación, con el mismo *mapping* del caso anterior. Esto representa un esquema de detección manual (que requiere acciones por parte del usuario), en el que todos los recursos de cómputo de ambos nodos del *cluster* se utilizan para tener ejecuciones redundantes, con posibilidad de comparar los resultados finales, posteriormente a la finalización de la ejecución (realizada como procesamiento *offline*); en caso de diferencia, es posible lanzar una tercera instancia, de forma de poder votar como resultado correcto el de dos de las ejecuciones que coincidan (al menos dos fallos en ejecuciones distintas deberían ocurrir para que no haya coincidencia). La diferencia de tiempo con la ejecución original se debe a la competencia por los recursos entre los procesos de las dos instancias. En caso de requerirse la tercera ejecución, sin modificar el *mapping*, se agregaría el tiempo de la primera fila (sin competencia por los recursos) al anterior. Sin embargo, esto sólo tiene en cuenta los tiempos de ejecución puros, sin contemplar el tiempo de comparación *offline* de los resultados, ni el de volver a lanzar la ejecución. Debe notarse que el tiempo total es independiente del instante de ocurrencia del fallo, debido a que, en cualquier caso, la ejecución debe concluir para poder verificar la validez de los resultados.

La tercera fila muestra el tiempo cuando se incorpora el mecanismo de detección de SEDAR. Se puede observar que el *overhead* introducido, debido a duplicación de procesos, la sincronización entre réplicas, la comparación y copia de los contenidos de los mensajes y la validación final de los resultados, es menor al 0.3%. En tanto, en caso de fallo, el tiempo total, que proviene de ejecutar hasta la detección, parar y relanzar desde el comienzo, es aproximadamente el doble del de la ejecución sin fallos, y representa un *overhead* aproximado del 6.5% respecto de realizar una tercera ejecución; sin embargo, SEDAR ofrece las ventajas de tener incluidos los tiempos de comparación y relanzamiento automático, que en el caso *offline* deberían ser llevados a cabo por el usuario. Además, debe notarse que este es el peor escenario posible, ya que todos los fallos inyectados son detectados casi sobre el final de la ejecución, como se explica a continuación. Por lo tanto, cuanto más tempranamente puedan ser detectados los fallos (menor latencia), mejor será el desempeño de la estrategia de detección y relanzamiento.

Tabla 1. Resultados de los experimentos

	Estrategia	**Tiempo [s] según caso de inyección**			
		Sin fallo	**Al comienzo**	**Intermedio**	**Al final**
1	**Sin SEDAR – 1 instancia**	329,57	-		
2	**Sin SEDAR – 2 instancias simultáneas**	373,16	702,73		
3	**SEDAR – det**	374,19	745,06	748,82	747,62
4	**SEDAR – rec (i = 60s); Nckpt = 6**	440,76	2066,23	1000,12	477,35
5	**SEDAR – rec (i = 90s); Nckpt = 4**	418,91	1569,85	822,93	459,63
6	**SEDAR – rec (i = 120s); Nckpt = 3**	407,06	1303,28	895,41	454,22

La situación mencionada se debe al comportamiento particular de la aplicación de prueba y su interacción con el mecanismo de detección. Como se mencionó en la sección 5.1, la multiplicación de matrices es fuertemente limitada por cómputo, y las fases de comunicación son breves y están ubicadas al comienzo y al final. Esto produce que, para todos los instantes de inyección seleccionados, la detección se produzca siempre durante la última comunicación colectiva. Como consecuencia, cuando el fallo se inyecta sobre el final (al finalizar el cómputo y antes de dicha comunicación), la latencia de detección es baja; pero cuando el fallo se inyecta antes de comenzar la fase de cómputo, o durante ella, permanece latente por mucho tiempo antes de ser detectado. En una aplicación con comunicaciones más frecuentes, la latencia de detección sería menor.

En la primera columna de las filas 4 a 6, se ven los tiempos correspondientes al agregado del mecanismo de recuperación, en ausencia de fallos. Se puede observar que el *overhead* es inversamente proporcional al intervalo de *checkpointing*. Este comportamiento es esperable, debido a que, si el intervalo es menor, aumenta el número de *checkpoints* y se consume más tiempo almacenándolos.

En tanto, el resto de las columnas muestra los tiempos de recuperación en presencia de los diversos casos de fallos. Puede notarse, en principio, que el mecanismo permite recuperar correctamente y finalizar la ejecución en todas las ocasiones, luego de distinta cantidad de reintentos según la latencia de detección. Sin embargo, sólo en el caso del fallo inyectado sobre el final, el tiempo de recuperación requerido es considerablemente menor que en la detección y relanzamiento automático desde el comienzo; pero en todos los demás, el almacenamiento de múltiples *checkpoints* para recuperar no representa una alternativa conveniente frente a simplemente detectar, parar y relanzar. Una vez más, este hecho está relacionado con la latencia de detección. Cuando la latencia es alta, los *checkpoints* que se almacenan mientras el fallo permanece latente están corruptos, y por lo tanto no son válidos para la recuperación. El algoritmo intenta recuperar sucesivamente hacia atrás desde cada uno de ellos, acumulando los tiempos involucrados en cada intento hasta encontrar un *checkpoint* válido[1]. En cambio, en el caso de los fallos inyectados sobre el final, que son rápidamente detectados, la recuperación desde el último *checkpoint* es exitosa, mejorando notablemente el tiempo total. Nuevamente, en una aplicación que realizara comunicaciones más frecuentes, la latencia de detección sería menor y, por lo tanto, también la cantidad de intentos de recuperación.

Como conclusión, se puede destacar la flexibilidad de SEDAR para ajustarse a las necesidades de un sistema particular, proveyendo diferentes posibilidades de cobertura (detección y relanzamiento automático o recuperación basada en múltiples *checkpoints*), que le permiten adaptarse para obtener un compromiso en la relación costo/beneficio. El intervalo de *checkpoint* óptimo y la frecuencia de validación de las comunicaciones pueden ser determinados a partir de la caracterización de la aplicación a proteger.

[1] Por ejemplo, en la inyección intermedia (durante el cómputo) con intervalo de *checkpointing* **i** = 60s, al momento de la detección se han realizado 6 *checkpoints*; la recuperación se intenta sucesivamente, sin éxito, desde el 6, luego desde el 5 y por último desde el 4, rehaciendo cada vez todo el cómputo hasta volver a detectar (y rehaciendo también los *checkpoints* que ya se habían hecho); recién es exitosa la recuperación desde el *checkpoint* 3.

6 Conclusiones y Trabajo Futuro

Garantizar la fiabilidad de las ejecuciones es uno de los principales desafíos en el camino hacia los próximos sistemas de cómputo en la Exa-escala. La protección de las aplicaciones MPI a nivel de las comunicaciones es un método factible y efectivo para detectar y aislar la corrupción de datos, evitando su propagación hacia todos los procesos que se comunican. En este trabajo, se presenta la herramienta SEDAR, que proporciona tres modos de utilización: sólo detección y relanzamiento automático, y recuperación basada en múltiples *checkpoints* de capa de sistema, ya sea periódicos o disparados por eventos. El comportamiento temporal de SEDAR en ausencia y en presencia de fallos, inyectados en distintos momentos durante la ejecución, permite evaluar su desempeño y caracterizar el *overhead* asociado a su utilización. La posibilidad de seleccionar entre las estrategias de detección y la de recuperación, y configurar el intervalo de *checkpointing*, proporciona flexibilidad para adaptar SEDAR a los requerimientos de cobertura y máximo *overhead* permitido de un sistema particular.

Entre los principales trabajos futuros, se destacan: la ampliación del trabajo experimental hacia aplicaciones con diferentes patrones de comunicación; la emulación de llamadas no determinísticas (requerida para extender el horizonte de aplicaciones que se pueden proteger con SEDAR); la determinación del intervalo óptimo de *checkpoint*; y la implementación de una estrategia de adaptación dinámica del mecanismo de recuperación a la ejecución en curso, basada en la determinación del momento a partir del cual vale la pena comenzar a proteger dicha ejecución mediante *checkpoints* (antes de ese instante, es más efectivo simplemente detectar y relanzar desde el comienzo. Como objetivo final, se busca la integración con arquitecturas que utilicen C/R para tolerancia a fallos permanentes [18], de modo de poder soportar ambos tipos de fallos con una única herramienta funcional para plataformas de la Exa-escala.

Referencias

1. T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and J. Labarta, “Fault-tolerant protocol for hybrid task-parallel message-passing applications," in 2015 IEEE International Conference on Cluster Computing , pp. 563-570.
2. A. Benoit, A. Cavelan, F. M. Ciorba, V. Le F_evre, and Y. Robert, “Combining checkpointing and replication for reliable execution of linear workflows with fail-stop and silent errors," International Journal of Networking and Computing, vol. 9, no. 1, pp. 2-27, 2019.
3. J. Elliott, M. Hoemmen, and F. Mueller, “Evaluating the impact of sdc on the gmres iterative solver," in Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, 2014, pp. 1193-1202.
4. D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, “Detection and correction of silent data corruption for large-scale high-performance computing," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012, p. 78.
5. A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, “PLR: A software approach to transient fault tolerance for multicore architectures," IEEE Transactions on Dependable and Secure Computing, vol. 6, no. 2, pp. 135-148, 2009.
6. H. Mushtaq, Z. Al-Ars, and K. Bertels, “Efficient software-based fault tolerance approach on multicore platforms," in Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 2013, pp. 921-926.
7. F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, “Toward exascale resilience: 2014 update," Supercomputing frontiers and innovations, vol. 1, no. 1, pp. 5-28, 2014.
8. G. Lu, Z. Zheng, and A. A. Chien, “When is multi-version checkpointing needed?" in Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale. ACM, 2013, pp. 49-56.
9. Montezanti, D., Frati, F.E., Rexachs, D., Luque, E., Naiouf, M.R., De Giusti, A.: SMCV: a Methodology for Detecting Transient Faults in Multicore Clusters.; CLEI Electron. J. 15(3), pp. 1-11 (2012).

10. Chen, Z.: Algorithm-based recovery for iterative methods without checkpointing. In Proceedings of the 20th international symposium on High performance distributed computing (pp. 73-84). ACM (2011).
11. Ferreira, K., Riesen, R., Oldfield, R., Stearley, J., Laros, J., Pedretti, K., & Brightwell, T.: rMPI: increasing fault resiliency in a message-passing environment. Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488 (2011).
12. Yalcin, G., Unsal, O. S., & Cristal, A.: Fault tolerance for multi-threaded applications by leveraging hardware transactional memory. InProceedings of the ACM International Conference on Computing Frontiers (p. 4). ACM (2013).
13. Engelmann, C., & Böhm, S.: Redundant execution of HPC applications with MR-MPI. In Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) (pp. 15-17) (2011).
14. Ni, X., Meneses, E., Jain, N., & Kalé, L. V.: ACR: Automatic checkpoint/restart for soft and hard error protection. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (p. 7). ACM (2013).
15. N. Ali, S. Krishnamoorthy, N. Govind, and B. Palmer, “A redundant communication approach to scalable fault tolerance in pgas programming models," in Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on. IEEE, 2011, pp. 24-31.
16. Montezanti, D., Rexachs, D., Rucci, E., Luque, E., Naiouf, M.R., De Giusti, A.: Characterizing a Detection Strategy for Transient Faults in HPC, in Computer Science & Technology Series, XXI Argentine Congress of Computer cience Selected Papers. ISBN 978-987-4127-00-6. La Plata. EDULP, 2016, pp. 77-90.
17. D. Montezanti, A. De Giusti, M. Naiouf, J. Villamayor, D. Rexachs, and E. Luque, “A methodology for soft errors detection and automatic recovery," in 2017 International Conference on High Performance Computing & Simulation (HPCS). IEEE, 2017, pp. 434-441.
18. M. Castro-León, H. Meyer, D. Rexachs, and E. Luque, “Fault tolerance at system level based on RADIC architecture," Journal of Parallel and Distributed Computing, vol. 86, pp. 98-111, 2015.
19. Benoit, A., Cavelan, A., Cappello, F., Raghavan, P., Robert, Y., & Sun, H. (2018). Coping with silent and fail-stop errors at scale by combining replication and checkpointing. Journal of Parallel and Distributed Computing, 122, 209-225.
20. F. Mathur and A. Avizienis, “Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair”, in Proc. Spring Joint Computer Conference (AFIPS '70), New York, USA, May. 1970, pp. 375 – 383.
21. Ansel, J., Arya, K., & Cooperman, G. (2009, May). DMTCP: Transparent checkpointing for cluster computations and the desktop. In 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-12). IEEE.